A livello piu` basso del kernel, ci stanno le SYSCALLS

rappresentano un transiente tra…

- **spazio utente**

e

- **spazio kernel**

L'apertura di un file, a livello utente, ad esempio, e` rappresentato proprio dall'uso di una syscall

- **SYS_OPEN**

oppure

- **__NR_open**

a livello kernel, per essere specifici.

Per una lista completa delle funzioni disponibili nel proprio sistema, basta dare uno sguardo al file:

- **/usr/src/linux/include/asm/unistd.h**

```
<--- Begin HERE --->

#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
This file contains the system call numbers.
*/

#define __NR_exit                        1
#define __NR_fork                        2
#define __NR_read                        3
#define __NR_write                       4
#define __NR_open                        5
#define __NR_close                       6
#define __NR_waitpid                     7
#define __NR_creat                       8
#define __NR_link                        9
#define __NR_unlink                     10
#define __NR_execve                     11
#define __NR_chdir                      12
.. .... .... .. . .. . . . . . .. .

.. .... .... .. . .. . . . . . .. .

<--- End HERE --->
```

In fase di init, il kernel tira su una tabella, un array.

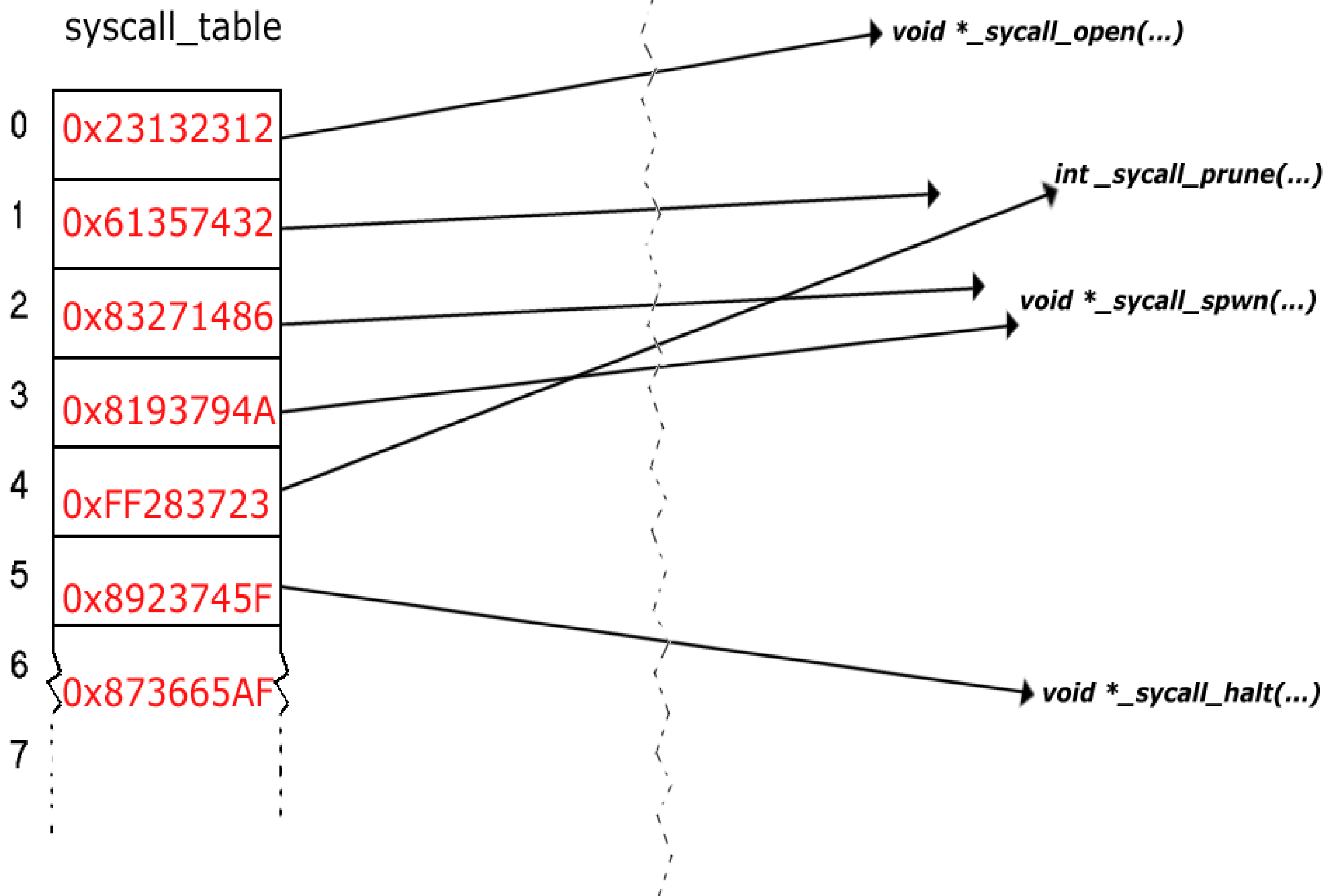
Dentro sono contenuti dei puntatori indicizzati da interi

- **Altro non è, che quello ke troviamo in**

unistd_H

quando e' necessario fare accesso ad una funzione…

…bisogna andare a scegliere, mediante l'indice, uno dei puntatori a funzione salvati nel nostro array.



quando e' necessario fare accesso ad una funzione…
…bisogna andare a scegliere, mediante l'indice, uno dei puntatori a funzione salvati nel nostro array

- ***dobbiamo sapere come muoverci e dove muoverci!***

- ***sapere cosa andare a toccare, e DOVE andarlo a toccare, e` fondamentale per la buona riuscita del nostro lavoro***

Esistono vari metodi (oltre l'esperienza), per sapere cosa andare a cercare

Vedi scorso workshop
e slides (quando saranno pubblicate)…

- **Un STRACE annesso a GDB del nostro ELF,**

od un

- **nm <object>**

ci daranno ampie informazioni su quello ke

il nostro codice in oggetto fa` …

…e con cosa lo fa`.

Un esempio: le SOCKETS

normalmente agiamo a livello di:

- **netfilter**

o di

- **packet_type**

quindi piu` vicini alle interfaccie di rete che alle chiamate dell'utente

(se ci si vuole ricondurre alla connessione bisogna controllare la porta, l'id del pacchetto, la sockaddr_in...)

Adesso lavoreremo in un modo piu` vicino all'utente che alle interfaccie…

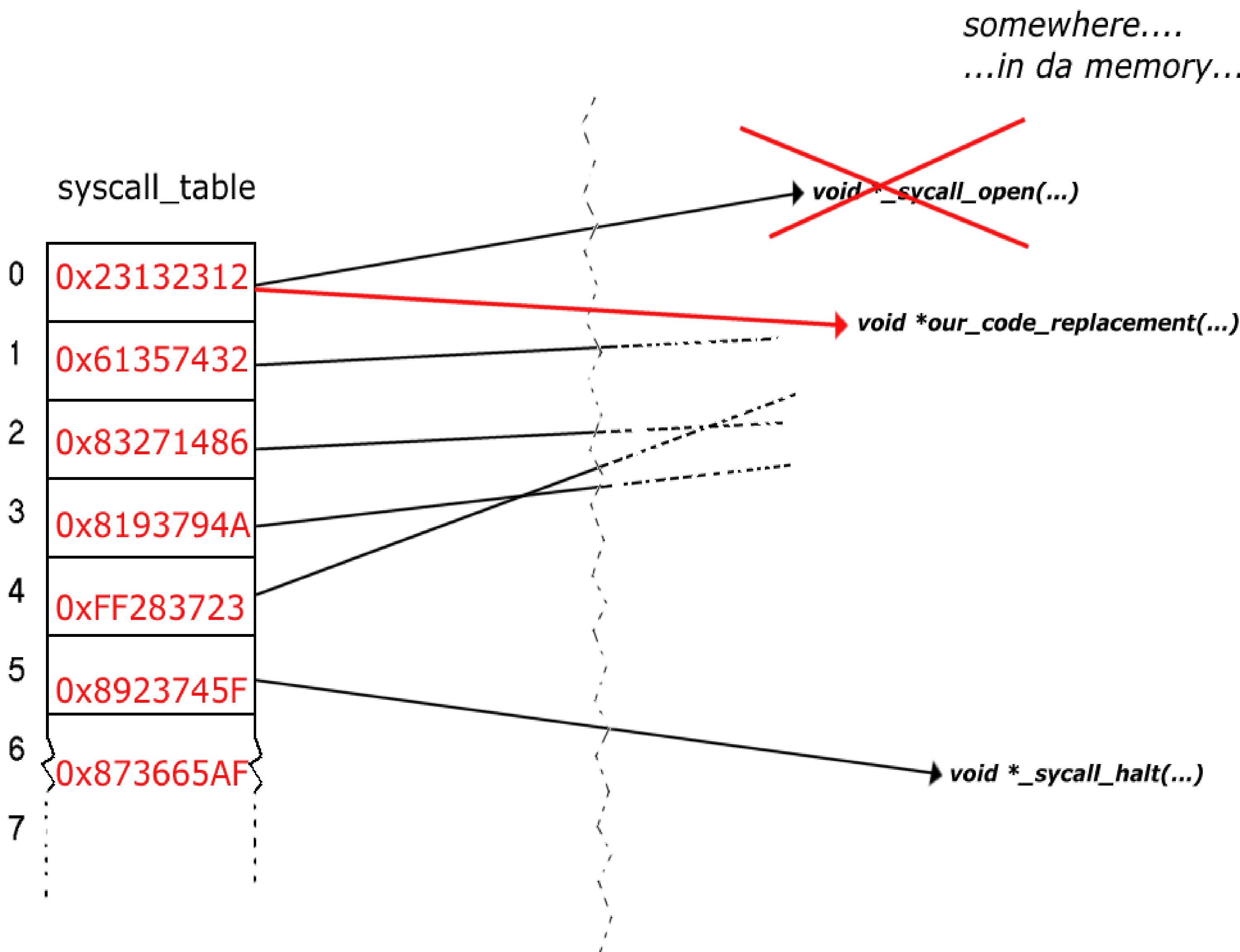
Per monitorare un certo demone od un certo programma

non ci servira` beccare i pacchetti e ricostruirli in relazione alla porta etc...

- **bastera` monitorare le chiamate bind**

(o la sys_bind o quelle di un protocollo che ci interessa)

a quel punto CAMBIARE le funzioni in relazione al lavoro ke intendiamo svolgere…



In realta` le cose non sono cosi` semplici!!

quello di cui abbiamo appena parlato è …

…solo una faccia della medaglia!

- **ci siamo occupati di come hookare le syscall**

ke hanno, un simbolo ESPORTATO nella

- **syscall_table**

puo' capitare di avere necessita` di…

- **wrappare, qualkosa ke non abbia un simbolo esportato**

od ancora

- ***che abbia solo un puntatore ad una serie di funzioni***

System.map

- **Durante l'installazione del nostro kernel viene creato un file:**

system.map

Il file System.map risulta utilissimo…

possiamo sapere l'esatta posizione in memoria di tutti i simboli

---
I simboli sono ogni variabile o funzione
dichiarata statica o globale
dentro a un file sorgente del kernel

- **quando apriamo /dev/kmem**
possiamo muoverci grazie a quegli offset

- **trovare proprio quei simboli**
- **tramite moduli usare indirizzi statici**

in modo da raggiungere punti del kernel

altrimenti *irraggiungibili…*

EXPORT_SYMBOL

- **abbiamo parlato di registrazione e occultamento dei simboli nei confronti di /proc/ksyms**

possiamo riguardare con occhio piu` critico a questo concetto…

E' possibile realizzare nella stesura del nostro codice una

tabella interna di simboli e operazioni

```
static struct symbol_table module_syms= {
    for_this: do_this,
    for_those:   do_those,
here:     make_this,
}
```

Quello ke viene fatto nella stesura di un modulo di kernel

- ***e' registrare questa tabella***

in maniera da ESPORTARE i simboli

- **EXPORT_SYMBOL(name)**
- **EXPORT_SYMBOL_NOVERS(name)**

Abbiamo parlato delle sockets, proprio per questo motivo

tra i simboli esportati non esiste __NR_socket

ma

__NR_socketcall

cosa significa?

- **__NR_socketcall è una chiamata di sistema ke wrappa la chiamata della struttura socket ke serve in quel particolare caso**

socket(...)

1. **il kernel effettuera` una chiamata a __NR_socketcall,**

2. **a sua volta, restituira` il controllo alla corretta funzione…**

Esiste una tecnica, largamente in uso da molti anni

- **descritta per la prima volta da Silvio Cesare**

si occupa proprio di come gestire questo tipo di eventualità…

quello ke viene fatto, non e` hookare

ma

andare a sovrascrivere la funzione di nostro interesse

Quello, ke vi illustrera` quest, altro non e` ke un

- **HIJACKING vero e proprio.**

In pratica si sovrascrivono i primi 7Bytes della funzione originale

(bastano 7bytes, per far quello ke ci serve per il nostro scopo)

con una porzione di codice del tipo:

```
asm {
    …
    mov offset della mia funzione, eax
    jmp eax
    ...
}
```

Quello che succedera` sara` qualkosa del tipo:

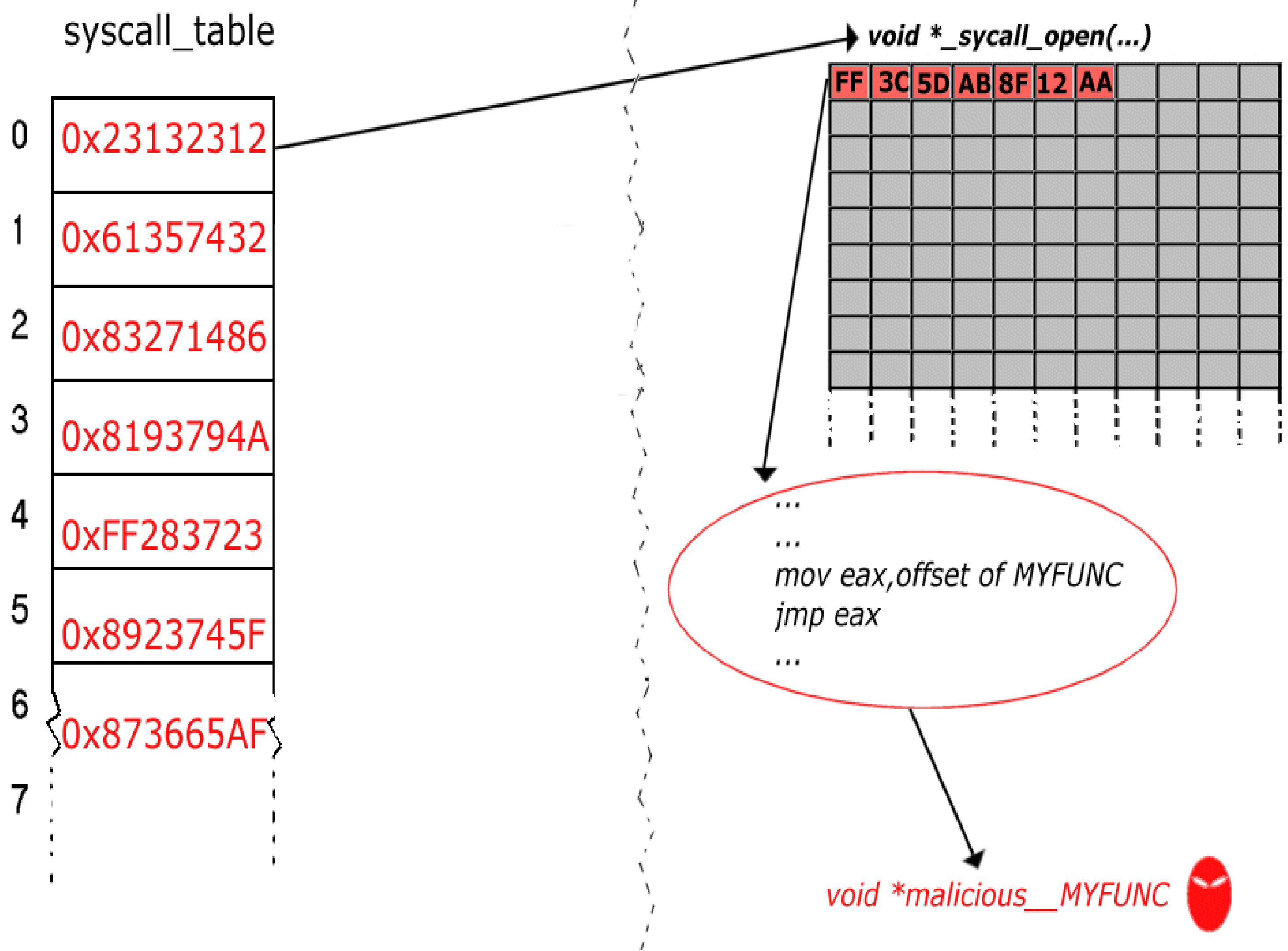
- **Effettuo una chiamata alla OPEN**

il puntatore mi rimanda all'indirizzo della static function ke gestisce l'apertura

in memoria, ci stanno i primi 7bytes della funzione riscritti, con una porzione di codice, che…

- **IMPONE un salto ad un'altra parte di memoria, dove ci stara` la MIA funzione…**

…il gioco e` fatto!


somewhere....
...in da memory...
syscall_table
0
0x23132312
1
0x61357432
2
0x83271486
3
0x8193794A
4
0xFF283723
5
0x8923745F
6
0x873665AF
7
void *_sycall_open(...)
FF
3C
5D
AB
8F
12
AA
...
...
mov eax,offset of MYFUNC
jmp eax
...
void *malicious__MYFUNC

Ovviamente, questo impone, alcuni accorgimenti ed accortezze

spiegai come conservare il puntatore alla nostra reale funzione prima della sostituzione con la funzione hajacked

necessario per un corretto ripristino della syscall_table

è necessario fare cosa analoga se viene usato il metodo di Cesare

- **In realtà, quì, la situazione è un pò più complessa**

Abbiamo SOVRASCRITTO 7bytes della funzione originale per poter stare al nostro gioco…

*quello ke viene fatto, che poi e' quello ke viene sfruttato, come idea, in un tool scritto da un caro amico (***kstat by fusys***)*

è quello di giocare su di un backup e monitoring dei primi 7bytes delle funzioni exported

Finora ci siamo concetrati sull'Hook delle syscalls

Abbiamo visti alcuni esempi di colme un attaccante possa operare in kernel space per occultarsi, rientrare, e sfruttare un sistema compromesso.

Ora vediamo come e' possibile "difendersi", nel senso di accorgersi che nel sistema, a kernel space qualcosa e' "cambiato".

Faremo cio' esaminando parte della suite KSTAT by FuSys

Cosa e' KSTAT?

DESCRIZIONE:

Kstat mostra delle informazioni prese direttametne dalle strutture del kernel. Questo e' realizzato tramite un accesso a /dev/kmem, quindi normalmente sono richiesti permerssi di root o di un utente che possa leggere kmem.

Questo e' specialmente utile quando noi non otteniamo dei risultati "validi" da usuali sorgenti ad applicazione, come dire, dopo un accesso non autorizzato al nostro sistema.

Kstat e' una tool che fa diversi tipi di controllo al fine di individuare delle possibili alterazioni di un fissato inprinting del sistema registrato al suo interno.

Noi non analizzeremo tutti gli aspetti di kstat, ci limiteremo a mostrarne due relativamente alle tecniche che abbiamo mostrato e che mostreremo, lasciando a voi come workAtHome di costatare tutte le altre e di adoperarvi per eluderle (= .

SYSCALLS HOOK

Abbiamo visto che qundo hookiamo una syscall noi non facciamo altro che sostituire un valore dentro una struttura dati: ***la sys_call_table[].***

Questo valore che noi adiamo a sostire altro non e' che l'*indirizzo* in kernel space a cui si trova "*fisicamente*" il codice binario compilato della nostra funzione.

Preso coscienza di cio' e' facilmente intuibile come andra' a lavorare kstat per effettuare un controllo sulla *sys_call_table[]:*

```
symex.c :

...
...

fd=fopen("include/syscall.h", "w");
fprintf(fd, "unsigned long calls[256]={");

if(kread(kd, (unsigned long)sys_call_table_addr,
   &syscall_table, sizeof(syscall_table)) == -1) {

     printf("kread error\n");
     exit(-1);
}

for(i=0; i<256; i++)
fprintf(fd, "%p,", (void*)syscall_table[i]);

fseek(fd, -1, 1);
fprintf(fd, "};\n");

...
...
```

Si crea in un include una copia della sys_call_table e poi :

syscalls.c

```
void show_syscalls(int replace)
{
    ...
    ...

    for(i=1; i < 256; i++)
        if(kmem_call_table[i]){
            if((unsigned long)kmem_call_table[i] != calls[i] &&
                    calls[i] != 0xffffffff && calls[i] != calls[0]){
                printf("\nsys_%-22s%16p", names[i],
                (void*)kmem_call_table[i]);
                printf(" WARNING! should be at %p",
(void*)calls[i]);
                w++;
            }
        }
        if(!w) printf("\nNo System Call Address Modified\n");
        printf("\n");
}
```

Verifica, quindi, che le entry della sys_call_table, letti direttametne da /dev/kmem siano congruenti con quelli che precedentemetne ha assunto come veritieri.

KERNEL HIJACKING

Fino ad ora abbiamo visto come **hookcare** una sys_call tramite la sostituzione di un in indirizzo dentro la *sys_call_table[],* e, come questo metodo di operare possa essere riconoscito. Introduciamo ora una tecnica sperimentata per la prima volta da *Silvio Cesare:* ***Kernel Function Hijacking.***

Essenzialmetne l'attacco dimostrato da Silvio si basa sull'idea di sostituire i primi byte di una funzione con una *asm jump* che salti, appunto, ad una funzione "*malicious*" inserita tramite un **LKM** in kernel space da un *attaccante* (= .

Un jump in asm e' un procedura del tipo:

```
movl $indirizzo_a_cui_saltare,%eax
jump *%eax
```

L'algoritmo:

In init_module:

- Salvare i primi 7 byte del codice della funzione originale per usarli successivamente.
- Settare il jump alla parte di codice da eseguire al posto della funzione originale.
- Sostituire i primi 7 byte della funzione originaria con quelli da noi predentemetne separati.

In cleanup_module:

- Ripristinare i byte della funzione originale con quelli che eravano stati salvati durante la init_module().

Nella funzione "malicious":

- Esegue qualcosa
  ...per chiamare al funzione orinaria...
- Rispristinare i 7 byte con quelli salvati precedentemente.
- Eseguire la funzione.
- Sostituire di nuovo i primi 7 byte con il jump.

CONCLUSIONI.

Ci sono altri metodi, piu' o meno proficui di utilizzare la tecnica di hijacking, un buon spunto, che mostra pure come andare a hijackare funzoni di cui non conosciamo direttamente l'indirizzo, ovvero il kernel non espoerta il simbolo, e' descritto da vecna in un articolo per *Bfi*.

Altri usi se ne possono fare, ma lasciamo a voi il divertimento di trovarli e sfruttarli*:* ***in kernel space non ci sono limiti alla fantansia…***

Catania 15 Aprile 2003
Valvoline & Quest